Non-linear Dynamics on the A_n^* Plane

Matthew Trefilek

April 2017

Table of Contents

1 Introduction

> Note: There may be some confusion rergarding the use of A* in regards to both the lattice representing the plane and the pathfinding algorithm. for the purposes of this paper, A_n^* represents the lattice and A* represents discussions about the pathfinding algorithm.

The goal of this project is to create a Java GUI tool that allows for the representation of non-linear dynamics in the A_n^* plane. The tool allows for the creation of attractors and repellers along with a series of "vehicles" that will navigate the space based on their specified targets. The paths that are created will represent the vector fields of the plane, and basins and peaks should appear around the attractors and repellers. It is the goal of this project to be able to abstract the code to work with a number of different uses.

2 A* Pathfinding

> Not all those who wander are lost.
>
> *J.R.R. Tolkien*

A* Pathfinding is a solution to the problem of finding the best path throughout some search area. In this case, the search area is the hexagonal lattice that makes up the 2-dimensional plane of our simulation. The optimal path will be the path that minimizes the function $f(n) = g(n) + h(n)$ for every node in the path. $g(n)$ represents the cost to get from the initial node to the current node. $h(n)$ is the heuristic function which estimates the cost to get from the current node to the end.

The heuristic in A* pathfinding is the estimated cost to move from any given node to the end node. There are a number of different functions that can be used to calculate $h(n)$. In the program, I used both a heuristic of $h(n) = 0$ and $h(n) = \Delta y + 2 * (\Delta x - 1)$ to calculate the pathfinding. When $h(n) = 0$, A* turns into Dijkstra's Algorithm. While not being as efficient, Dijkstra's promises to return the optimal path. Listing 1 shows the example code using this $h(n)$. This second $h(n)$ allows for a calculated heuristic close

to what the path would be assuming a regular set of hexagons with the same weight. A* with a heuristic is more effective than Dijkstra's because A* can eliminate nodes who have a large $f(n) = g(n) + h(n)$ faster than when $f(n) = g(n)$. However, if $h(n) > f(n)$, A* will sometimes return a less optimal path but at a faster rate.

```
while (!openNodes.isEmpty()) {
    Hexagon current = minimumFScore(openNodes, fScore);
    current.setColor(Color.GREEN);
    if(current.equals(goal))
        return createPath(cameFrom, current);

    openNodes.remove(current);
    alreadyVisited.add(current);
    for(Hexagon neighbor : current.getNeighbors()) {
        if(alreadyVisited.contains(neighbor))
            continue;

        double tGScore = gScore.get(current) + neighbor.getWeight();
        if(!openNodes.contains(neighbor))
            openNodes.add(neighbor);
        else if (tGScore >= gScore.get(neighbor))
            continue;

        cameFrom.put(neighbor, current);
        gScore.put(neighbor, tGScore);
        fScore.put(neighbor, 0.0);
    }

}
```

Listing 1: Sample Java Code of A* Pathfinding with $h(n) = 0$

3 Non-linear Dynamics

The study of Non-linear Dynamics is a subset of Chaos Theory, the theory that certain processes cannot be modelled but only estimated. Non-linear dynamics studies the change within a state-space (the space in which we are observing) over some set of parameters (time, temperature, etc). Once the state space has a number of trajectories existing within it, a vector field can be constructed from the differentiation of these paths. Once the vector field has been created, a trajectory can be estimated once given a starting position. Non-linear dynamics offers a number of advantages in modelling real world behavior, however one of the most prominent is the continued modification of the vector field. After a sufficient number of trajectories have been observed, a vector field can be created for modelling purposes. However, if anything about the state space changes, the system model can be updated with new trajectory data that is collected.

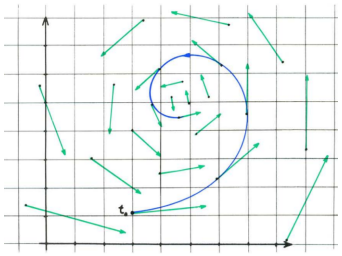


Figure 1: A vector field created from the differentiation of the trajectories of the state space

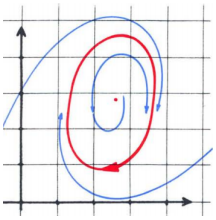


Figure 2: A vector field with an attractor curve and repeller point

While the study of non-linear dynamics can goes much deeper than simple vector fields, the focus of the project was utilizing the concepts of repellers and attractors. An attractor is a point or curve that is the center or limit of a subset of the vector field. A repeller, is the opposite where the vector field limits in the opposite direction. When looking at the trajectories on the state space, attractors tend to create valleys or troughs where the trajectories flow "down," and the repellers create peaks as the trajectories flow away from the center of the repeller. Figure 2 shows an example of a cyclic vector field where the trajectories enter into the area below the attractor curve and are

then repelled back up away from the repeller point.[1]

4 A_n^* Lattice

> Aggregates form a nested sequence of tessellations of n-space.
>
> Kitto et al.

4.1 Permutahedron

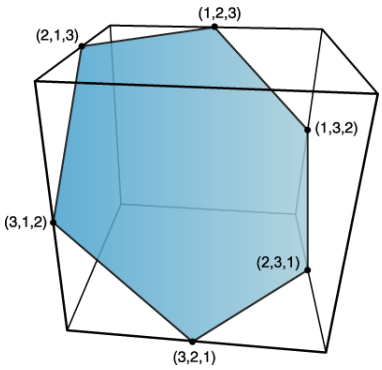


Figure 3: The second order permutahedron bisecting the square.

A permutahedron is geometric object of flat sides. The permutahedron of order n contains $n!$ vertices and exists in the n-dimensional space. The first order permutahedron can be thought of the point on a line. The second order permutahedron can be thought of as the line spanning from (1,2) and (2,1). The third order permutahedron represents the hexagon that bisects the cube with side length 2. This pattern continues up to infinity with the dimensions of both the shape and the object it exists within increasing. For the purposes of this program we will be focusing on the standard hexagon, or the permutahedron with order 2. For the sake of simplicity, the coordinate system will be rotated so the viewer is orthogonal to the center of the hexagon, resulting in a 2-dimensional plane.

4.2 Aggregates and the A_n^* Lattice

A_n^* represents the lattice made up of the tiling of a space with permutahedra of order n. The tiling of a space with hexagons can be constructed in several

[1] Abraham; Shaw. Dynamics: The Geometry of Behavior. Addison-Wesley Publishing Company. 1992

different ways. This tiling, commonly known as a honeycomb lattice, has several advantages discussed later, however it runs into challenges with the accessing of cells and traversal of the lattice. Kitto, Vincem and Wilson in the 1991 paper "An isomorphism between the p-adic integers and a ring associated with a tiling of N-space by permutohedra" provide new ways for fast addressing and retrieval of cells in the lattice. In A_2^*, an addressing system of cells can be created using the *generalized balance ternary* (GBT_2) where the address is a sequence $s_1 s_2 ... s_k$ where $0 \leq s_i \leq 6$ $i = 1...k$. This address is known as the canonical address. The canonical address, which is in $\mathbb{Z}/7\mathbb{Z}$ can converted to a standard address, presented as a series of vectors within $\mathbb{Z}/2\mathbb{Z}$. The following example represents the hexagon with canonical address 342 as a standard address:

$$342 = \begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}$$

To understand the addressing system graphically, the important thing in the construction of the lattice is the aggregate. Figure 4 shows the aggregates of levels from 0–3 with examples of canonical addressing within the aggregate level. The highlighted section represents the $n-1$ central aggregate. To navigate between aggregates, we define the $(n+1) \times (n+1)$ matrix B:

$$B = \begin{pmatrix} 2 & 0 & \cdots & 0 & -1 \\ -1 & 2 & \cdots & 0 & 0 \\ 0 & 0 & \cdots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & -1 & 2 \end{pmatrix} \qquad B_2 = \begin{pmatrix} 2 & 0 & -1 \\ -1 & 2 & 0 \\ 0 & -1 & 2 \end{pmatrix}$$

The inverse to this matrix B is given by:

$$B^{-1} = (1/q)(b_{ij}) \qquad b_{ij} = \begin{Bmatrix} 2^{n+j-i} & i \geq j \\ 2^{j-i-1} & i < j \end{Bmatrix} \qquad q = 2^{n+1} - 1$$

To get the cell $x \in A_n^*$ using the canonical address $s_1 s_2 ... s_k$ you must first convert the canonical address into the standard address $t_0 t_1 ... t_{k-1}$. Then:

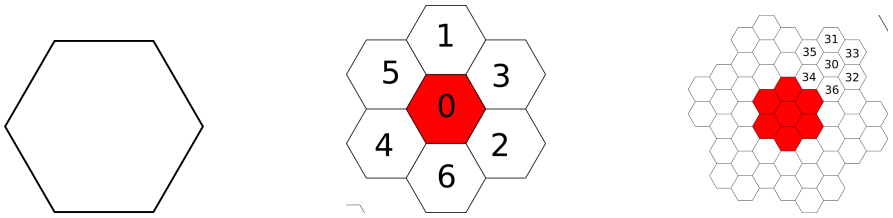
$$x = V(t_0 + B_n t_1 + B_n^2 t_2 + \cdots + B_n^{k-1} t_{k-1})$$

where if V is the $n \times (n+1)$ matrix whose columns $v_0, v_1, ..., v_n$ generate the lattice A_n^*, the result is the vector equation that gives the cell x. The

example given by Kitto et al shows the process of finding the location of the cell 362 in A_2^*:

$$x = a_0v_0 + a_1v_1 + a_2v_2 \qquad \begin{pmatrix} a_0 \\ a_1 \\ a_2 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix} + B_2 \begin{pmatrix} 0 \\ 1 \\ 1 \end{pmatrix} + B_2^2 \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}$$

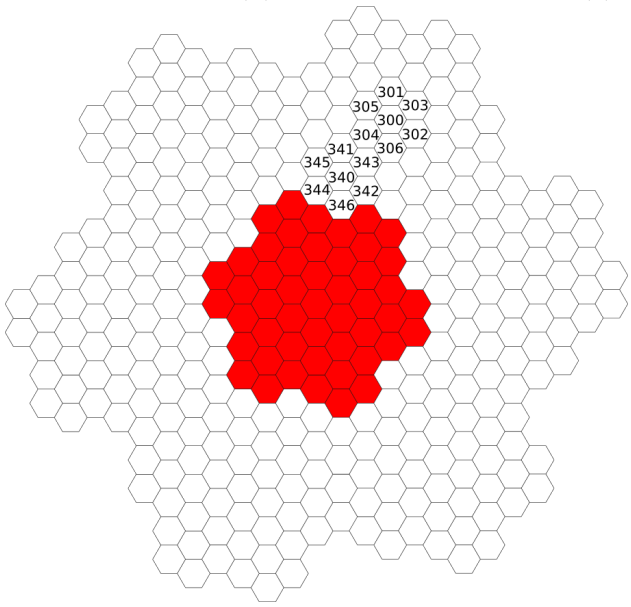
Which gives $x = 1 * v_0 + 7 * v_1 + -3 * v_2$. Using B_n^{-1} it is possible to take the vector equation of a cell and get the standard address.[2]



(a) Level 0

(b) Level 1

(c) Level 2



(d) Level 3

Figure 4: Aggregates at levels {0, 1, 2, 3} with example addressing

[2]Kitto; Vince; Wilson. An isomorphism between the p-adic integers and a ring associated with a tiling of N-space by permutohedra. The University of Florida, Gainesville. 1991.

4.3 Why Hexagons?

Hexagons were chosen for this application because they offer a number of advantages over other geometric shapes. First, the lattice, commonly referred to as the honeycomb lattice, can completely cover a plane with no overlap or other geometric shapes. Secondly, the hexagon has more edges than a square, which allows it to closer approximate a circle. In the real world, there are infinitely many directions that can be moved from a point. Although the hexagon is still bounded by 6, it is more effective at this modelling than a square grid. Finally, a hexagon has 6 equidistant neighbors, that is the length of the line between the center of the hexagon to the center of any of its 6 neighbors is equivalent for all of the neighbors.

Another reason for the choice of hexagons is their applications in image processing. One of the long-term goals of this model is to be able to use it to process image data. The reasons presented above make a good case for their effectiveness, but also that "biological and opthalmic observations on the human eye indicate that a hexagonal packing of retinal sensory elements has evolved in nature. This was a motivation for the study of hexagonal sampling schemes for computer vision covered in this chapter."[3]

5 Automata

A common field of study in both robotics and computer science is the concept of an automaton, a self moving or navigating item. Originally in the planning of the project, I called the items navigating the hexagonal lattice Cellular Automata. I originally thought that that would be what I was working with because cellular automata are often thought of in relation to some sort of plane. However, the vehicles that I was dealing with more closely resembled Braitenberg vehicles.

A Braitenberg vehicle is a vehicle named after the Italian-Austrian Cyberneticist Valentino Braitenberg. "In *Vehicles*, Braitenberg describes a set of though experiments in which increasingly complex vehicles are built from simple mechanical and electronic components. Each of these imaginary vehicles in some way mimics intelligent behavior."[4] Braitenberg's thinking was

[3]Staunton. Hexagonal Sampling in Image Processing. University of Warwick Coventry. 1999.

[4]Hogg; Martin; Resnick. Braitenberg Creatures. MIT Media Laboratory. June 5, 1991.

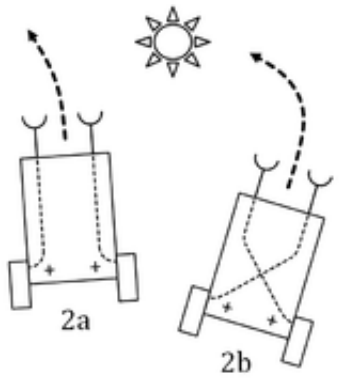


Figure 5: Examples of two simple Braitenberg vehicle

that from the interaction between sensors and mechanics, intelligent behavior would emerge. Figure 5 shows two examples of the simple vehicles with motors on the two back wheels and sensors on the front. The sensors in these pick up the strength of the light emitting from the light source and control their attached motor which will either cause the light to attract or repel the vehicle.

Similar to these simple vehicles the vehicles in my program rely on their ability to sense the weight of the hexes around them when making their movement choice. Similar to the attraction and repulsion qualities of the light, the low troughs and high spikes of the hexagons have similar effects on the motion of the automata navigating this plane.

6 Implementation

6.1 Graph-based Approach

After running into some issues with the mathematics behind the A_n^* lattice I set up the world similar to a graph. Each hexagon has 6 neighbors which are stored in an array. The SquareWorld implementation was what this became to be known. The SquareWorld is constructed with a number of rows and columns and is indexed from the upper left going row by row. The address for a hexagon at (x, y) can be found with the equation $index = y*\text{COLUMNS}+x$ where COLUMNS is the number of columns of the world. This approach

works well and is easy to comprehend and implement because it is similar to the usual Cartesian Coordinate System. However, it suffers from slower traversals by needing to rely of the neighbors instead of vectors produced within the before mentioned lattice. Having the SquareWorld also does not take advantage of the aggregate tiling that makes the lattice much more scalable while preserving the representation of a circle that many programs use hexagons for.

However, this approach did have it's upsides. While vector based lookup was not as easy to implement, being a graph made the A* Pathfinding Algorithm very easy to implement and it worked very well with navigating between the attractors and repellers. The attractors and repellers were created using a recursive algorithm which expanded to the hex neighbors using the function:

$$w(x) = 1 + 2 * \frac{tan^{-1}(.1 * x)}{\pi}$$

Where $w(x)$ is the new weight of the hexagon where x is the integer weight of the hex. This function was chosen, as it is a map from $(-\infty, \infty)$ to $(0, 2)$. As each ring of hexes is affected, $x = log_2(x)$. So the weight of each ring follows a logarithmic decay. To produce an attractor, the weight is set through $w(-x)$ so that attractor weights are less than 1 while repeller weights are greater than 1 with every hexagon having a default weight of 1.

6.2 Utilizing the Lattice

The hardest part of the program, the implementation of the A_n^* lattice, was the part that took the most time, and that is still not entirely complete. The lattice class currently has methods for the conversion between the standard and canonical addresses. After creating a matrix class for dealing with the vector and matrix calculations of the space, it is possible to convert between the standard address to the vector representation of the cell and back again. Unfortunately, the issues that are arising have to do with the display portion. After drawing the Level 1 aggregate, I am having trouble translating the lattice vectors back into a coordinate system that can be used for drawing the successive vectors. This coordinate mapping is something that would greatly increase the speed of the software and is something I will be attempting to complete the week before the presentation.

7 Coding Practices

7.1 Tools

Java

I chose Java for this project for a number of reasons. Java is an object oriented language and offers easy implementation of interfaces and abstract classes, which I took advantage of. It offers a number of pre-built libraries, specifically graphics ones, that I was interested in utilizing. I also wanted to work with Java as it is the language much of my work for my new job will be in. Finally, Java GUI frameworks are all cross platform which was important since I was developing on both Windows and Linux.

JavaFX

After researching and creating an initial GUI with Java Swing, I ended up switching to JavaFX for a number of reasons. I chose FX primarily for the vector drawing canvas capabilities. The switch to FX required a lot of new learning, but it ended up being very efficient. As the successor to swing, FX relies on a Model View Controller architecture. I found FX did a lot better job than swing at abstracting the view and controller code. Similar to Android layout files, the view lies in a .fxml file, an xml stylesheet that has callbacks to methods in a specified controller file. This abstraction allows for the user to interact with the data model and for the controller to update the view separately.

```
<Button text="Save Image" onAction="#handleSaveButtonAction"
 ↪  fx:id="savebutton" focusTraversable="false"/>

@FXML private Button savebutton;

@FXML private void handleSaveButtonAction(){}
```

Listing 2: Sample Button in JavaFX

In JavaFX, the FXML file contains the layout information. As seen in Listing 2 the onAction tag refers to the @FXML method name within the

controller. The button object can be accessed through the @FXML button declaration. Another advantage to JavaFX is the ability to capture the state of the Canvas as an image. This ability provided an easy way to save and view tests that were run.

JUnit

For the unit testing, I used JUnit4. JUnit offers a strong test suite for writing unit tests. I wrote tests for all of the specialty classes I created. I also tested edge cases, particularly object comparisons and pathfinding issues. JUnit4 offers a number of tools such as setup and teardown for both all tests and individual tests. As shown in Figure 6 JUnit4 and IntellijIDEA work very well together to run and review tests. The tests were also the most robust and complete I had ever written, covering close to 150 lines of code. Listing 3 shows two example unit tests.

```
1  @Test
2  public void testHashMap(){
3      DefaultHashMap<Integer, Integer> map = new
       ↪  DefaultHashMap<>(Integer.MAX_VALUE);
4      Assert.assertEquals((Integer)map.get(new Integer(4)),
       ↪  (Integer)Integer.MAX_VALUE);
5  }
6  @Test
7  public void testNullHex(){
8      Hexagon nullHex1 = Hexagon.nullHexagon();
9      Hexagon nullHex2 = Hexagon.nullHexagon();
10
11     Assert.assertEquals(nullHex1.isNull(), true);
12     Assert.assertTrue(nullHex1.equals(nullHex2));
13     Assert.assertEquals((double)nullHex1.getWeight(), Double.MAX_VALUE,
       ↪  0);
14
15 }
```

Listing 3: Example from selected Unit Tests

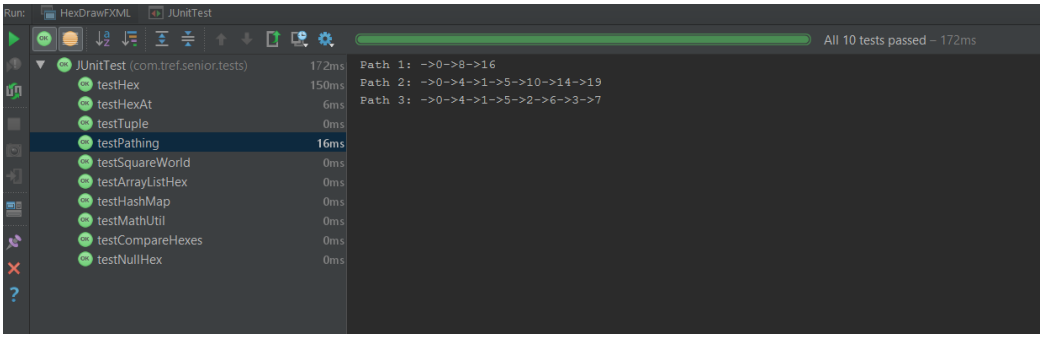


Figure 6: JUnit Testing within Intellij

7.2 Project Structure

I focused on creating a well structured object-oriented program. I also wanted to keep my methods lengths down. While I did not hold myself strictly to the 10-lines or less rule, I did try my best to pull code into a new method if it was getting too long. The final result of this was much cleaner looking code. In the end, the code was some of the best that I had ever written. I held true to Object Oriented Design, the MVC architecture, and a number of various patterns.

Another effort I made was to treat the project like an industry project. I tried to effectively utilize git, especially while working from different devices. I focused on creating helpful and informational commit messages, and I only ever pushed when all unit tests passed. I kept all development away from the master branch, and after each merge ran all unit tests again. A tool I used to help with tasking was Trello, a free kanban style task board. Although I was the only "employee" I used Trello to manage upcoming tasks and features.

7.3 Coding Patterns

Below are a few code samples from some specific patterns represented.

Singleton and Null Object Pattern

```
private static Hexagon nullHex;
private Hexagon() {
    isNull=true;
}
public static Hexagon nullHexagon(){
    if(nullHex == null) {
        nullHex = new Hexagon();
    }
    return nullHex;
}
```

Listing 4: Example of the Null Object and the Singleton Patterns combined

Visitor Pattern

The visitor pattern was used for both drawing with Java Swing and JavaFX and for printing information to the terminal.

```
@Override
public void visit(Hexagon h) {
    System.out.print(h.getSquareIndex());
}

@Override
public void visit(SquareWorld w) {
    for(Hexagon h : w.getHexes())
        h.accept(this);
}
```

Listing 5: Print visitor example

Observer Pattern

The Observer Pattern played a large part in the MVC design that I was following. When the model, the hexagonal world, updated, it called notified it's observers, the view which then ran the appropriate visitor to produce the

output. Java makes the Observer Pattern extremely easy to implement with the built-in Observer and Observable classes.

```
@Override
public void update(Observable o, Object arg) {
    redraw();
}
private void redraw() {
    clear();
    gc.scale(scale.getK(),scale.getT());
    scale = new Tuple<>(1.0,1.0);
    visitorFXML.visit(world);
}
```

Listing 6: Observer Pattern Example

8 Personal Outcomes

8.1 Computer Science

Over the course of this project, I have grown strongly in three areas. First, in my knowledge of Java and what it has to offer. I had much preferred C++ for Object Oriented Programming up to this point. However, my work with Java has made me respect it much more. I have gotten used to utilizing interfaces and abstract classes in Java. I have also gotten better manipulating instances of classes without needing to use pointers directly. As my job I will be starting uses a lot of Java, I am glad I chose Java as the language of choice.

Secondly, I have gained much more experience with UI design. Although the UI is rather simple for this project, my research into both Swing and JavaFX taught me a lot about MVC design and techniques. I learned how to properly access items in a view and how to manipulate the data. Finally, in reading the documentation I grew in my knowledge of UI best practices, how to structure and format the user experience.

Thirdly, I was able to work on a project as if I was working with a team. Although I worked on my own, I focused more on design methodology and

unit testing than I ever have outside of class. I effectively used git and unit tests to create a good code work flow. My extra research into the patterns and design methodology has definitely made me a better programmer.

8.2 Math

One of the largest outcomes for me was the experience of applying my knowledge from advanced mathematics classes to a real application. I had seen Abstract Algebra applied in several other ways during my classes. However, this project gave me the chance to utilize Abstract Algebra in my computer science class. It was very clear in the programming that the difference between the graph based approach and the mathematical lookup approach was huge. Being able to access a specific address in constant time along with converting between an address and a vector of direction saves a lot of computation time, and allows for easier construction of the plane.

An outcome for both Computer Science and Math has been my use of LaTeXfor the writing of this paper and also the presentation. Although I have used LaTeXbefore, I got to explore a large amount of packages, including Beamer for the presentation and Tikz for drawing mathematical shapes. As I hope to continue research in both Math and Computer Science, I am glad that my knowledge of LaTeXis improving.

9 Conclusion

Overall, I am very happy with the project. I succeeded in gaining a large amount of information from the research that I conducted. I also feel that the code I had written was some of the best I ever had. I held true to the patterns that I was implementing. Although I will make some changes before I finish on Friday, the project currently totals 1400 lines of actual code not counting blank lines or comments. The program fulfills many of the goals I set forth. The tool is able to create a Hexagonal plane and run automata pathfinding simulations on it.

I am planning on continuing to improve the tool, completing the implementation of the more advanced lattice addressing. Once the base tool is done, the data within the hexes can be changed allowing for a number of different applications. For example, an image can be loaded in and pixel values represented by the data within the hexagons. As the automata crawl the

plane, they will naturally avoid the steeper gradients between colors. The produced vector field would represent repellers along the edges of the image.

List of Source Codes

List of Figures

Figure Sources

Figure 1: Abraham; Shaw. Dynamics: The Geometry of Behavior. Addison-Wesley PublishingCompany. 1992.
Figure 2: Abraham et al.
Figure 3: Permutahedron. wikipedia.org. 2017.
Figure 5: Braitenberg Vehicles. wikipedia.org. 2017.